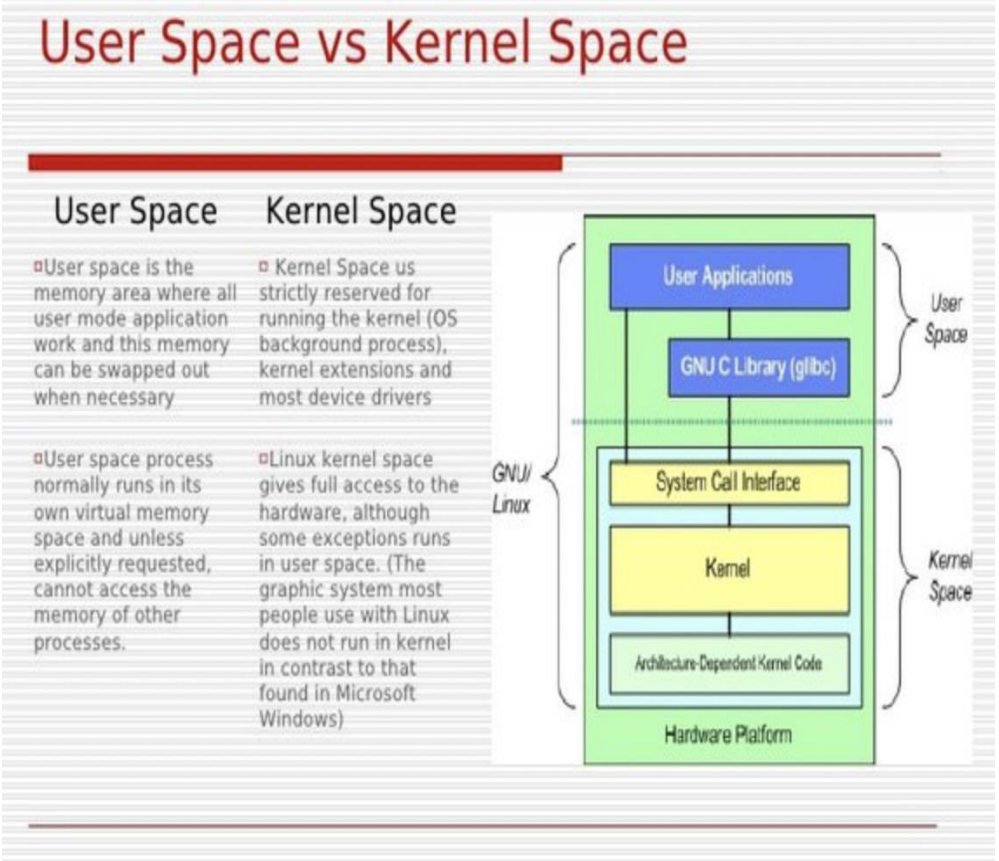
TOPICS

- User Space/Kernel Space
- System Calls
- Basics of Pointer

User Space Vs Kernel Space

- **Kernel space** is where the kernel (i.e., the core of the operating system) runs and provides its *services*. It's something that the user is not allowed to interfere with.

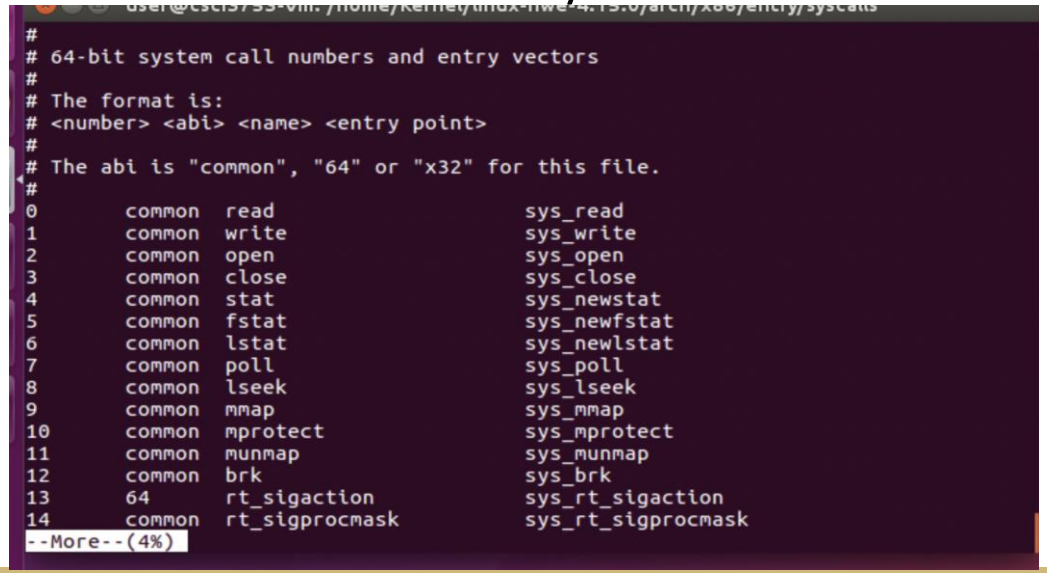
- **User Space** is that portion of system memory in which *user processes* runs.



System Calls

- ➔ A programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on.
- ➔ Each system call corresponds to a number defined in a syscalls table.

- ➔ **syscall(number, ...)**



Types of system calls:

There are 5 different categories of system calls

1. Process control: end, abort, create, terminate, allocate and free memory.
2. File management: create, open, close, delete, read file etc.
3. Device management
4. Information maintenance
5. Communication

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()

Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Do we use system calls at all?

- **Example 1:**

- If you open a file using fopen() in the library stdio.h, it gets translated into the **open()** system call.

- In the standard library, the user-space implementation of the open() system call executes and passes the system call number.

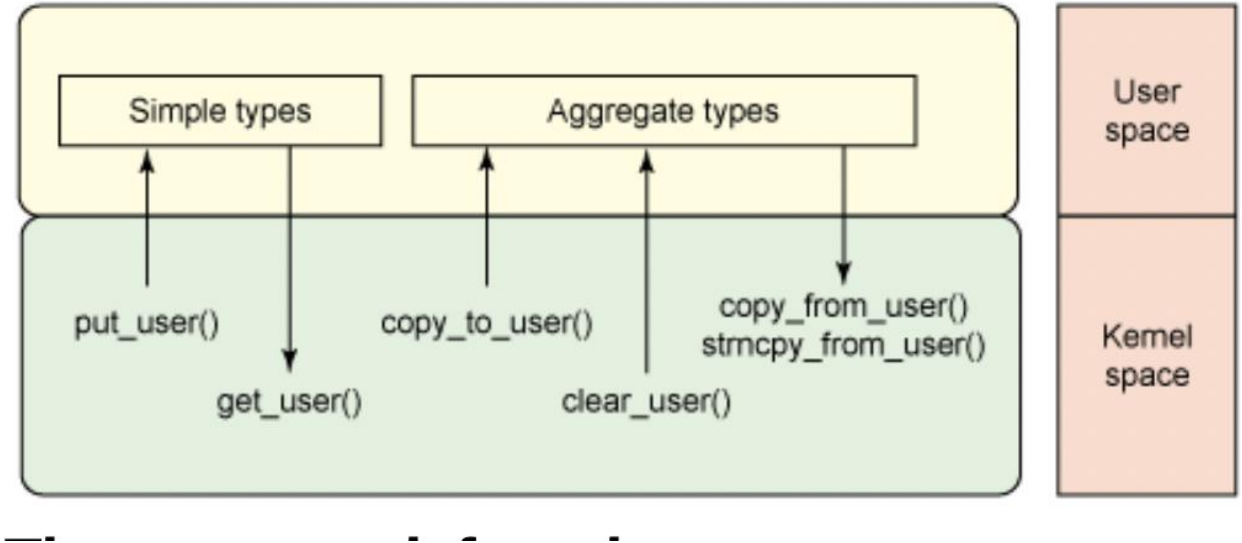
- **Example 2:**

- In Unix-like systems, fork() and exec() are C- library functions that in turn execute instructions that invoke the **fork()** and **exec()** system calls.

Executing a System Call

1. A user space program invokes the syscall
2. A (typically) software interrupt called a trap is triggered (INT)
3. Mode bit is flipped from user to kernel (1 to 0).
4. The interrupt tells the kernel which syscall was called.
    a. Requisite data may be passed in
    b. The kernel verifies if all parameters are legal before executing the system call
5. After execution,mode bit flips and user program resumes

Data Movement Between user space and kernel Space



The User Space Memory Access API

Function	Description
`access_ok`	Checks the validity of the user space memory pointer
`get_user`	Gets a simple variable from user space
`put_user`	Puts a simple variable to user space
`clear_user`	Clears, or zeros, a block in user space
`copy_to_user`	Copies a block of data from the kernel to user space
`copy_from_user`	Copies a block of data from user space to the kernel
`strnlen_user`	Gets the size of a string buffer in user space
`strncpy_from_user`	Copies a string from user space into the kernel

Why can't you just call, say, memcpy?

Two reasons:

1. The kernel is capable of writing to any memory. User process's can't. **copy_to_user** needs to check **dst** to ensure it is accessible and writable by the current process.
2. Depending on the architecture, you can't simply copy data from kernel to user-space. You might need to do some special setup first, invalidate certain caches, or use special operations.

Pointers

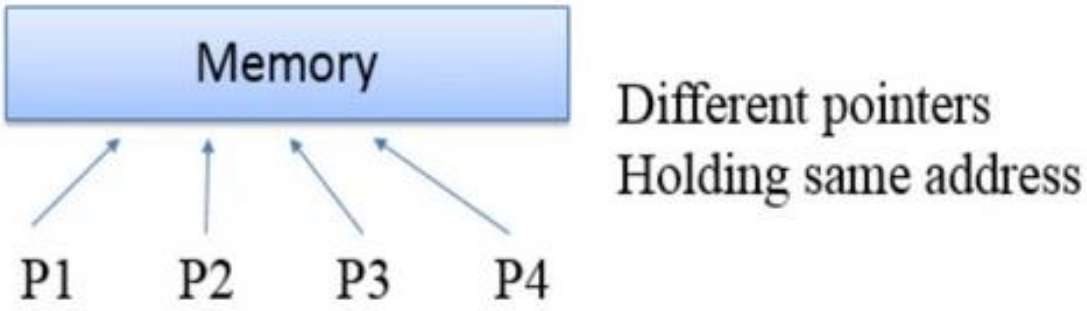
- A variable that holds a memory address
- This address is the location of other object

Exception:

Null pointer : Doesn’t contain any address and it’s value is 0

Pointer can have 3 types of content associated with:

1. Address
2. Null
3. Invalid values (Doesn’t correspond to any memory location)



If you modify/delete the memory location, same is reflected in all pointers

Declaring pointer

- Data-type * name
- * is a unary operator, called dereference operator
- * is used to declare the pointer and also to dereference it(Finding the value at that particular memory location)

int * ptr;

The variable “ptr” stores a pointer to an “int”

Pointer operations in c

- Creation : Returns variable’s memory address
- Dereference : Returns contents stored at address
- Indirect Assignment : Stores value at address
  - *pointer = value
- Direct Assignment : Stores pointer in another variable
  - Pointer1 = pointer2

Simple c program:

```
int x;
int *p;
x = 0;
p = &x;
*p = 1;
```

What will be the value of x?

Exercise 1:

```
Int i1 = 1;

Int i2 = 2;

Int *ptr1;

Int *ptr2;

Ptr1 = &i1;

Ptr2 = ptr1;


*ptr1 =3

I2 = *ptr2;
```

Exercise 2:

```
Int i = 100;
Int j = 200;
Int *ptr1 = &i;
Int *ptr2 = &j;
Ptr1 = *ptr2;
Ptr1 = ptr2;
```

Valid?

What's the value of i and j?

Pass by reference

```
Void setvalues(int *x, int *y) {
        *x = 100;
        *y = 200;
}
Void callfunction(void) {
        x = 1;
        y = 2;
        setvalues(&x, &y);
}
```

Extra topics for fun:

- Pointer for strings
- Pointer Arithmetic
- Pointer to Pointer

Thank You!